Chapter 12
Data Structure

Associate Prof. Yuh-Shyan Chen
Dept. of Computer Science and Information Engineering
National Chung-Cheng University

Outline

12.1 Introduction
12.2 Self-Referential Structures
12.3 Dynamic Memory Allocation
12.4 Linked Lists
12.5 Stacks
12.6 Queues
12.7 Trees

12.1 Introduction

- Dynamic data structures - grow and shrink during execution
  - Fixed-size data structure (single-subscripted array, double-subscripted array)
- Linked lists (Linear) - insertions and removals made anywhere
- Stacks (Linear) - insertions and removals made only at top of stack
- Queues (Linear) - insertions made at the back and removals made from the front
- Binary trees (Non-linear) - high-speed searching and sorting of data and efficient elimination of duplicate data items

12.2 Self-Referential Structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a NULL pointer (0)

- Two self-referential structure objects linked together

- Data member and pointer
- NULL pointer (points to nothing)
12.2 Self-Referential Classes (II)

```c
struct node {
    int data;
    struct node *nextPtr;
}
```

- `nextPtr` - points to an object of type `node`
- Referred to as a link – ties one node to another node

12.3 Dynamic Memory Allocation

- Dynamic memory allocation
- Obtain and release memory during execution

```c
malloc
```
- Takes number of bytes to allocate
- Use `sizeof` to determine the size of an object
- Returns pointer of type `void *`
- `void *` (pointer to void) to the allocated memory.
- If no memory available, returns `NULL`
- `newPtr = malloc( sizeof( struct node ) );`

```c
free
```
- Deallocates memory allocated by `malloc`
- Takes a pointer as an argument
- `free (newPtr);`

12.4 Linked Lists

- Linked list
  - Linear collection of self-referential class objects, called nodes, connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member
  - Link pointer in the last node is set to `null` to mark the list’s end

- Use a linked list instead of an array when
  - Number of data elements is unpredictable

An example of linked list

```c
struct node {
    int data;
    struct node *nextPtr;
}
```
12.4 Linked Lists (II)

- **Types of linked lists:**
  - **singly linked list**
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction
  - **circular, singly linked**
    - Pointer in the last node points back to the first node
  - **doubly linked list**
    - Two "start pointers": first element and last element
    - Each node has a forward pointer and a backward pointer
    - Allows traversals both forwards and backwards
  - **circular, doubly linked list**
    - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

---

**Main program**

```c
int main()
{
    int choice;

    char ListNode::startPtr = NULL;

    int listIMpty();
    void insert();
    char choice;
    char item;

    printf("Choice: ");
    scanf(\"%d\", &choice);

    switch (choice) {
        case 1:
            printf("Enter a character: ");
            scanf(\"%c\", &item);
            insert(item);
            listIMpty();
            break;
    }
}
```

---

**Inserting and deleting node in a list**

```c
struct ListNode { /* self-referential structure */
    char data;
    struct ListNode *nextPtr;
};

typedef struct ListNode ListNode;

void insert(ListNode **startPtr, char ch);
void delete(ListNode **startPtr, char ch);
int listIMpty(ListNode **startPtr);
void printList(ListNode **startPtr);
```

---

```c
/* Fig. 12.3 list03.c */

int main()
{
    int choice;

    char ListNode::startPtr = NULL;

    int listIMpty();
    void insert();
    char choice;
    char item;

    printf("Choice: ");
    scanf(\"%d\", &choice);

    switch (choice) {
        case 1:
            printf("Enter a character: ");
            scanf(\"%c\", &item);
            insert(item);
            listIMpty();
            break;
    }
}
```

---

```c
/* Fig. 12.3 list03.c */

int main()
{
    int choice;

    char ListNode::startPtr = NULL;

    int listIMpty();
    void insert();
    char choice;
    char item;

    printf("Choice: ");
    scanf(\"%d\", &choice);

    switch (choice) {
        case 1:
            printf("Enter a character: ");
            scanf(\"%c\", &item);
            insert(item);
            listIMpty();
            break;
    }
}
```
Function instructions

```plaintext
// Print the instructions
void instructions(void)
{
    printf("Select one of the following:
    1 to insert an element into the list
    2 to delete an element from the list
    3 to end.
    ");
}
```

Inserting a node in order in a list

```plaintext
void insertNode(int, char[] node)
{
    int data = (int)node;
    int position = 0;
    while (list[position].data < data)
    {
        position++;
    }
    list[position].next = list[position + 1];
    list[position].data = data;
    list[position + 1].next = NULL;
}
```
Deleting a node from a list

Print out a list

Deleting a node from a list

Print out a list
12.5 Stacks

- Stack
  - New nodes can be added and removed only at the top
  - Similar to a pile of dishes
  - Last-in, first-out (LIFO)
  - Bottom of stack indicated by a link member to NULL
  - Constrained version of a linked list
- `push`
  - Adds a new node to the top of the stack
- `pop`
  - Removes a node from the top
  - Stores the popped value
  - Returns `true` if `pop` was successful

Graphical representation of a stack

A simple stack program

```c
// Fig. 12.4: fig12_04.c
/* dynamic stack program */
#include <stdlib.h>
#include <stdio.h>

typedef struct stackNode { /* null-reference structure */
  int data;
  struct stackNode *nextPtr;
} stackNode;

void push(stackPtr *, int);
int is_empty(stackPtr);

int main(void)
{
  stackPtr stack;
  int x;

  printf("Enter number of stack operations: ");
  scanf("%d", &x);

  for (int i = 0; i < x; i++)
  {
    char op;
    printf("Enter operation (push or pop): ");
    scanf("%c", &op);

    if (op == 'p' || op == 'P')
    {
      int value;
      printf("Enter value: ");
      scanf("%d", &value);
      push(&stack, value);
    }
    else if (op == 'p' || op == 'P')
    {
      int poppedValue = pop(&stack);
      if (poppedValue == -1)
        printf("Stack is empty.
    
      int main(void)
```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program

? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL

? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program

? 3
End of run.
Function `pop`

```c
83
84
85
86
87
88
89
90
91
92
93
94
95
96
int pop(StackNodePtr *topPtr)
{
tempPtr = *topPtr;
povValue = (*(topPtr))->data;
*topPtr = (*(topPtr))->nextPtr;
return povValue;
}
```

Pop operation

```
(a) *topPtr   12   7   11
(b) *topPtr   12   7   11
```

Applications of Stack

- Stacks support recursive function calls
12.6 Queues

- Queue
  - Similar to a supermarket checkout line
  - First-in, first-out (FIFO)
  - Nodes are removed only from the head
  - Nodes are inserted only at the tail

- Insert and remove operations
  - Enqueue (insert) and dequeue (remove)

- Useful in computing
  - Print spooling, packets in networks, file server requests

Example of queue operation

```c
/* Fig. 12.13: fig12.13.c
Operating and maintaining a queue */
#include <stdio.h>
#include <stdlib.h>

struct queueNode { /* self-referential structure */
   char data;
   struct queueNode *nextPtr;
};
typedef struct queueNode queueNode;
typedef QueueNode *QueueNodePtr;

// function prototypes
void printQueue( QueueNodePtr );
char dequeue( QueueNodePtr * , QueueNodePtr * );
void enqueue( QueueNodePtr * , QueueNodePtr * , char );
void instructions( void );
```
enqueue function

```c
void enqueue(QueueNode **headPtr, QueueNode **tailPtr, char value)
{
    QueueNode *newPtr = malloc(sizeof(QueueNode));
    if (newPtr == NULL)
        return;
    newPtr->data = value;
    newPtr->nextPtr = *tailPtr;
    if (*tailPtr)
        (*tailPtr)->nextPtr = newPtr;
    *tailPtr = newPtr;
    if (*headPtr) (*headPtr) = newPtr;
    else
        printf("No not inserted. No memory available.",value);
}
```

dequeue function

```c
char dequeue(QueueNode **headPtr, QueueNode **tailPtr)
{
    QueueNode tempPtr;
    QueueNode *tempNode;
    *headPtr = (*headPtr)->nextPtr;
    value = (*headPtr)->data;
    tempNode = *headPtr;
    *headPtr = *headPtr->nextPtr;
    free(tempPtr);
    return value;
}
```
12.7 Trees

- Tree nodes contain two or more links
  - All other data structures we have discussed only contain one
- Binary trees
  - All nodes contain two links (connecting with left and right child nodes)
  - None, one, or both of which may be NULL
  - The root node is the first node in a tree.
  - Each link in the root node refers to a child
  - A node with no children is called a leaf node
12.7 Trees (II)

- Binary search tree
- Values in left subtree less than parent
- Values in right subtree greater than parent
- Facilitates duplicate elimination
- Fast searches - maximum of \( \log n \) comparisons

```
struct treeNode {
    struct treeNode *leftPtr;
    int data;
    struct treeNode *rightPtr;
};

typedef struct treeNode TreeNode;

treeNode *createNode(int value){
    TreeNode *newNode = (TreeNode*)malloc(sizeof(TreeNode));
    if(newNode == NULL)
        return NULL;
    newNode->data = value;
    newNode->leftPtr = newNode->rightPtr = NULL;
    return newNode;
}
```

```
void insertNode(TreeNode *treePtr, int value){
    if(treePtr == NULL)
        return;
    if(value < treePtr->data)
        insertNode(treePtr->leftPtr, value);
    else if(value > treePtr->data)
        insertNode(treePtr->rightPtr, value);
    else
        printf("Value already exists!");
}
```

```
main function
```
12.7 Trees (III)

- **Inorder traversal**: prints the node values in ascending order
  1. Traverse the left subtree with an inorder traversal.
  2. Process the value in the node (i.e., print the node value).
  3. Traverse the right subtree with an inorder traversal.

- **Preorder traversal**:
  1. Process the value in the node.
  2. Traverse the left subtree with a preorder traversal.
  3. Traverse the right subtree with a preorder traversal.

- **Postorder traversal**:
  1. Traverse the left subtree with a postorder traversal.
  2. Traverse the right subtree with a postorder traversal.
  3. Process the value in the node.

```
Example of a binary search tree

13 47
25 77
11 43 65 93
7 17 31 44 68
```

The preOrder traversal is:
27 13 6 17 42 33 48

The inOrder traversal is:
6 13 17 27 33 42 48

The postOrder traversal is:
6 17 13 33 48 42 27

```
36 void inOrder( TreeNodePtr treePtr )
37 { 
38   if ( treePtr != NULL ) {
39     inOrder( treePtr->leftPtr );
40     printf( "%d", treePtr->data );
41     inOrder( treePtr->rightPtr );
42   }
43 }
44
45 void preOrder( TreeNodePtr treePtr )
46 { 
47   if ( treePtr != NULL ) {
48     printf( "%d", treePtr->data );
49     preOrder( treePtr->leftPtr );
50     preOrder( treePtr->rightPtr );
51   }
52 }
53
54 void postOrder( TreeNodePtr treePtr )
55 { 
56   if ( treePtr != NULL ) {
57     postOrder( treePtr->leftPtr );
58     postOrder( treePtr->rightPtr );
59     printf( "%d", treePtr->data );
60   }
61 }
```